VŠB - TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2013 **Martin Dorazil**

VŠB - TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
KATEDRA INFORMATIKY

Vývoj subsystému herního enginu pro RTS

Development of Game Engine Subsystems for RTS

2013 **Martin Dorazil**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Dorazil**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vývoj subsystémů herního enginu pro RTS
Development of Game Engine Subsystems for RTS

Zásady pro vypracování:

Cílem práce je vytvoření několika subsystémů herního enginu pro hry typu FPS. Při realizaci zadání se snažte volit co nejefektivnější postupy a algoritmy. Implementaci proveďte v jazyce C++ a dodržujte Google C++ Style Guide. Grafický subsystém bude založen na OpenGL. Zvažte také možnost využití aplikačních akcelerátorů typu NVIDIA PhysX pro vytvoření fyzikálně korektního prostředí. V textu práce popište zejména algoritmy a metody, které jste použil při řešení zadaných oblastí.

1. Seznamte se s typickými architekturami herních enginů.
2. Navrhněte následující subsystémy: generování prostředí, vzájemné kolize objektů, tzv. occlusion culling, jednoduchá AI.
3. Výsledky průběžně kombinujte s paralelně vyvíjenými moduly.
4. Funkčnost celého systému ověřte na jednoduchém demu, pro které vytvořte vhodné modely.

Seznam doporučené odborné literatury:

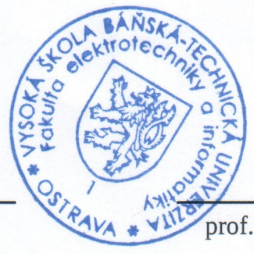
[1] Gregory, J. Game Engine Architecture. 2009. ISBN 9781568814131.
[2] McShaffry, M. Game Coding Complete. Third edition. 2009. ISBN 1584506806.
[3] Ericson, C. Real-Time Collision Detection. 2005. ISBN 9781558607323.
[4] Millington, I., Funge, J. Artificial Intelligence for Games. Second edition. 2009. ISBN 9780123747310.
[5] Wright, R. S., Lipchak, B., Haemel, N. S. OpenGL(R) SuperBible: Comprehensive Tutorial and Reference. Fifth edition. 2010. ISBN 0321712617.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6.5.2013

Podpis: .....

Rád bych poděkoval panu Ing. Tomáši Fabiánovi za pomoc a vedení při tvorbě této práce. A dále Pavlu Balcárkovi za vývoj subsystémů rozšiřujících výslednou aplikaci.

Abstrakt

Práce se zabývá různými typy herních enginů, se zaměřením na jejich grafickou část. Dále pak konkrétněji enginy pro hry typu RTS, problematiku generování terénu, tvorbu entit umístěných v tomto terénu, interakce mezi nimi, ale také využití možností OpenGL pro tvorbu reálně vypadajícího prostředí. Výsledkem implementace je výkonný a stabilní základ pro herní engine, s možností vygenerování jedinečného prostření obsahujícího pohyblivé i nepohyblivé entity.

Herní engine, RTS, OpenGl, generování terénu

Abstract

This thesis deals with different types of game engines, with a focus on the graphics subsystem. Furthermore, particularly engines for games such as RTS, problems generating terrain, creation entities located in this field, the interaction between them, but also the possibility of using OpenGL for creating real-looking environment. The result of the implementation is powerful and stable basis for the game engine, with the possibility of generating a unique game enviroment comprising movable and unmovable entities.

Game engine, RTS, OpenGl, terrain generation

Seznam použitých zkratek

FPS	- First Person Shooter
RTS	- Real-Time Strategy
GLSL	- OpenGL Shading Language
AI	- Artificial intelligence
SDL	- Simple DirectMedia Layer
VBO	- Vertex Buffer Object
MVP	- Model View Projection (matice)
GUI	- Graphical User Interface
AABB	- Axis-Aligned Bounding Box
OBB	- Oriented Bounding Box

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Vývoj počítačových her jde stále kupředu stejně, jako vývoj technologií používaných v této oblasti. Nezbytnou součástí moderní počítačové hry je výkonný grafický engine, na tuto problematiku je prioritně zaměřena tato práce. Dále se zabývá architekturou herních enginů, metodami procedurální tvorby herního prostředí, ale také metodami jak urychlit vykreslování scény apod. Cílem této práce je vytvořit stabilní, výkonný herní engine s použitím moderních technologií a postupů. pro tvorbu grafického subsystému je použito OpenGL, pro správu událostí, multithreading a ošetření uživatelských vstupů knihovny SDL.

Ve druhé kapitole najdeme definici pojmu herní engine, jeho rozdělení, ale také dělení podle typů video her. Seznámení se s hrami typu RTS. Jsou zde uvedeny informace o použitém hardwaru a softwaru, na kterém byla testována přiložená demo aplikace.

Třetí kapitola se zabývá problematikou vykreslování scény. Tato kapitola je nejrozsáhlejší. Najdeme zde krátké seznámení s knihovnou OpenGL a dále pak použití nejnovějších technologií vykreslování. Popis VBO, implementaci a také možnost použití indexování. Základní možnosti shaderů a jejich implementace v tomto enginu, ukázky zdrojových kódů vertex a fragment shaderů. Metody jak přiblížit vzhled scény realitě, tedy postup vytváření reflexe a stínů v tomto enginu. Jako poslední v této kapitole je uveden průběh tvorby celé scény.

Čtvrá kapitola je zaměřena na architekturu herních enginů a zvláště pak na metody použité v tomto projektu. Najdeme zde rozdělení herní mapy na jednotlivé části a detailní popis jejich funkce. Způsob přístupu k datům je v této kapitole také popsán. Jako ukázka je uvedena sekvence načítání shader programů.

Pátá kapitola obsahuje postupy pro procedurální generování prostředí použitého v tomto enginu. Zaměřena je zvláště na generovaní terénu. Najdeme zde vysvětlení funkce diamod-square algoritmu a také možnosti jeho kombinace s Voroného diagramy pro dosažení co možná nejvíce reálného vzhledu terénu.

V šesté kapitole najdeme řešení kolizí mezi entitami na mapě. Definici pojmu bounding box, pohled na jeho použití v tomto enginu a také jeho rozdělení na více typů. Dále zde nalezneme použité metody pro detekci kolizí a jejich popis.

Sedmá kapitola se zabývá metodami jak zrychlit vykreslování scény. Obsahuje popis několika způsobů jak odstranit zpracování částí terénu, ale i objektů, které nejsou při určitém pohledu na scénu viditelné.

2 Herní engine a jeho subsystémy

2.1 Herní engine

Pod pojmem herní engine (game engine) myslíme systém pro realizaci video her. Jedná se o jakési jádro video her, které můžeme rozdělit na několik dalších částí (subsystémů) řešících konkrétní oblasti dané problematiky. Pojem "game engine"vznikl v polovině 90. let v souvislosti s hrami typu FPS (first-person shooter) jako je např. populární Doom vyvinutý společností id Software, kde je již dobře patrná hranice mezi komponentami jádra systému a grafickou částí, herním světem atd.

2.2 Dělení na subsystémy

Herní engine můžeme rozdělit na tyto subsystémy:

- Vykreslování (rendering engine)
- Fyzika nebo detekce kolizí
- Zvuk
- Animace
- Skriptování
- AI (artificial intelligence)
- Síťová vrstva

Námi vyvinutý engine, se kterým se dále seznámíme v této práci, zahrnuje správu vykreslování, detekci kolizí, jednoduché animace, skriptování a AI.

2.3 Dělení podle typu her

Dnešní video hry můžeme mimo jiné dělit podle jejich typu a to takto:

- First-Person Shooters (FPS)
- Bojové hry (Fighting Games)
- Závodní hry (Racing Games)
- Real-Time Strategy (RTS)
- Massively Multiplayer Online Games (MMOG)

S tímto dělením se liší také požadavky kladené na herní engine. My se dále budeme zabývat hrami typu RTS a to zejména jejich grafickou částí.

2.4 RTS

Za první moderní RTS (Real-Time Strategy) video hru můžeme považovat Dune II vyvinutou v roce 1992 společností Westwood Studios, jejímiž následovníky jsou dnes velice populární hry Warcraft, Age of Empires, Starcraft a další. Hry tohoto typu poskytují hráči rozsáhlé herní pole (mapu), kde si buduje svou základnu ze zdrojů nabytých těžbou surovin. Cílem je vytvořit dostatek válečných jednotek na získání převahy nad protihráčem. Herní svět je typicky zobrazen v pohledu shora, kdy vidíme velkou plochu určité herní oblasti.

Starší hry byly často založeny na organizaci jednotlivých částí herní plochy do mřížky, což zjednodušovalo a usnadňovalo renderovaní. Používaly ortografickou projekci (konkrétněji izometrickou projekci) což je metoda pro vizuální reprezentaci 3D objektů ve 2D, ve které úhly mezi osami x, y a z jsou stejné a to 120°. Tato metoda zobrazení je použita např. v klasické hře Age of Empires.

Moderní RTS hry často používají realistickou perspektivní projekci skutečného 3D světa, ale stále používají mřížku pro usnadnění organizace objektů (budov, jednotek apod.) na herní ploše. Tímto případem je i náš game engine.

2.5 Problematika řešená v této práci

Hlavním cílem je implementace grafického subsystému s použitím knihoven OpenGL s podporou shaderů a vertex buffer objectů (konkrétní verze uvedeny dále). Námi implementovaný subsystém se bude pokoušet o vytvoření co nejreálnějšího herního prostředí, obsahujícího generovaný terén, vodní hladinu, oblohu, ale také vegetaci a pohybující se objekty.

Zaměříme se také na tvorbu realistických efektů jako jsou vržené stíny, odraz objektů na vodní hladině, deformace objektů a ploch pod hladinou vody, vše s použitím GLSL shaderů a OpenGL.

Každý herní grafický subsystém se pokouší o vykreslení často komplikované scény v co nejkratším čase, aby nebyla narušena plynulost hry. Existuje celá řada metod jak zrychlit celý proces renderu scény, na některé z nich se také zaměříme. Výsledný engine by tedy měl být schopen vykreslovat poměrně složitou scénu v přijatelném čase a nepřetěžovat grafickou kartu zbytečnými daty.

Velice důležité je při implementaci herního enginu dobře navrhnout jeho strukturu, zejména u rozsáhlejších projektů (což vzhledem ke komplexnosti herního enginu může být i tento projekt) je třeba myslet dopředu a najít účinná a přehledná rešení daných problémů. Špatný návrh může postupem času a růstem projektu způsobovat jeho nestabilitu, obtížné hledání vzniklých chyb případně snížení až znemožnění jeho rozšiřování, bez toho aniž by musel být z velké části přepsán. I tato problematika bude dále řešena.

U RTS her je kladen velký důraz na herní logiku, často je herní systém velice komplikovaný např. hry řady Settlers (vyvíjené společností Blue Byte od roku 1993 dodnes) poskytují komplexní strukturu získávání a zpracování zdrojů, jednotky mají své profese a jsou řízeny logikou enginu (nikoliv samotným hráčem jako tomu je u her řady Age of Empires). Nedílnou součástí herní logiky moderních enginů je i možnost skriptování, které je mimo jiné dobře použitelné právě na logiku hry, AI jednotek apod.

2.6 Hardware a software

Testováno na:

- CPU Intel Core i5-2430M 2,40 GHz
- RAM 8 GB
- GPU AMD Radeon HD 6630M (požadována podpora minimálně OpenGL 3.3)

Software a hlavní použité knihovny:

- Windows 7 64 bit
- Ubuntu 12.04 LTS amd64
- OpenGL 4.2.11566
- GLSL 420
- SDL 1.2.15

Celý projekt byl vytvářen v Eclipse 3.7.2 / Visual Studiu 2010 a kompilován pomocí kompilátoru gcc / Visual C++. Další použité knihovny: glew-1.9.0, glm-0.9.4.2, boost-1.53.0, lua a luabind. Demo aplikace, přiložená na DVD k této práci, je zkompilovaná pomocí gcc a testována na Ubuntu 12.04 LTS amd64.

2.7 Další informace

Modely a textury použité v ukázkové aplikaci jsou zdarma dostupné na `thefree3dmodels.com`, byly na nich provedeny pouze drobné úpravy v mapování textur a jejich velikosti. Do budoucna však nejsou vhodné pro další použití v tomto enginu. Jsou učeny pro detailní zobrazení například ve hrách typu FPS a podobných, což pro RTS není z hlediska množství viditelných objektů vhodné.

3 OpenGL

OpenGL (Open Graphics Library) je API (Application Programming Interface) pro tvorbu interaktivních 2D a 3D aplikací. Od počátku vývoje v roce 1992 se stalo velice populárním a široce rozšířeným na velkém množství platforem. Poskytuje vysokou vizuální kvalitu a zároveň velkou rychlost zobrazovaní scény. Společně s knihovnami SDL (Simple DirectMedia Layer), které jsou mimo jiné v tomto projektu také použity, se stává velice silným nástrojem pro tvorbu 3D video her.

OpenGL 3.x

Největším rozdílem mezi stále používaným OpenGL 2.x (a staršími) a OpenGL 3.x je absence fixní pipeline. Neexistuje zde žádný vestavěný světelný model, jednoduchá správa textur ani možnost okamžitého odesílání vertex dat. Proto probíhá modernizace a přechod na novější verze OpenGL pomalu, zvláště u komplikovaných aplikací postavených na fixní pipeline ([2]). Mnoho tutoriálů a knih popisuje použití fixní pipeline. Z použití OpenGL 3.x také vyplývá nevýhoda, že pro vše musí být napsány příslušné shader programy (viz. kapitola 3.2).

3.1 Vertex Buffer Objekty (VBO)

Jedná se o mechanismus pro přesun geometrie a textur do grafické karty. Tato data nejsou vykreslena okamžitě, ale až při volání funkce *glDraw*. Jedním z problémů, při vykreslování, je přesun dat do grafického akcelerátoru. Většinou je akcelerátor připojen k hostitelskému systému pomocí vysokorychlostní sběrnice. Pokaždé když chceme data vykreslit, musí nejprve být přesunuta z aplikační paměti a zpracována pomocí *pipeline* (obr. 1). S použitím VBO je možné data před zpracováním přenést blíže k akcelerátoru, přímo do paměti grafické karty. Toto řešení umožňuje výrazné zrychlení celého vykreslovacího procesu a umožní tak plné využití předností dané grafické karty.

Implementace

VBO je třeba vytvořit pouze jednou při inicializaci, dále postačuje uchovávat jeho číselný identifikátor. V tomoto enginu, každý objekt ve scéně má svůj VBO. Terén je rozdělen do několika VBO v návaznosti na dělení scény (viz. kapitola 7.4). U entit je logika inicializace a použití VBO naimplementována ve třídě *Model*. Díky struktuře *poolů*, zprostředkovávající přístup k datům (viz. kapitola 4.6), je pro jeden model vytvořen vždy jeden VBO, který může být následně použit při vykreslování více entit stejného typu.

Nový VBO je vytvářen pomocí *glGenBuffers* po načtení geometrie modelu, *glBindBuffer* připraví vytvořený buffer k použití a *glBufferData* nahraje příslušnou geometrii do VBO. Pro samotné vertexy, normály a UV koordináty, načteného modelu, jsou vytvářeny oddělené VBO. Tedy každý model má 3 VBO. Pro vykreslení objektu je ve třídě *Model* naimplementována funkce *Draw*. V této funkci je pro každý ze tří VBO třeba nejprve pomocí funkce *glEnableVertexAttribArray* povolen příslušný layout pro předání dat do shaderů (viz. kapitola 3.4). Dále je nabindován příslušný VBO pomocí *glBindBuffer* a funkce *glVertexAttribPointer* nastaví formát v jakém předáváme data. Tento postup je třeba aplikovat pro VBO vektorů, normál a UV koordinátů. Pomocí funkce *glDrawArrays* je model vykreslen.

Indexování

Indexování je jednou z metod jak zmenšit množství zpracovávaných dat. Ve 3D grafice skládáme objekty z trojúhelníkových sítí, jeden trojúhelník tvoří vždy 3 vertexy, ke kterém jsou přidružena i další

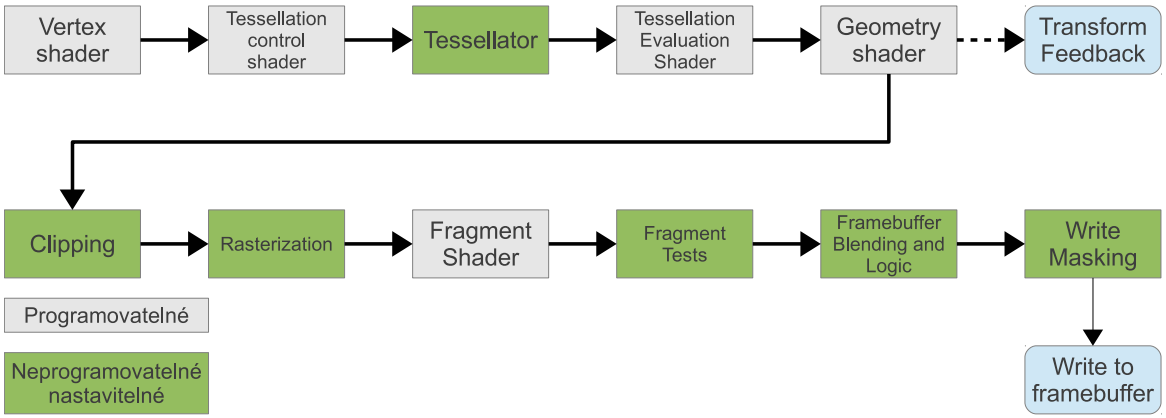
data, jako normály a UV koordináty pro texturování. V dříve popsaném postupu vykreslní pomocí *glDrawArrays* a bez zařazení indexování, vzniká duplicita bodů, pro každý z trojúhelníků jsou vytvořeny vždy 3 nové body. Algoritmus pro indexování prochází před samotným nahráním dat do VBO všechny vertexy a zkoumá jejích duplicity. Dva vartexy jsou totožné pokud: mají stejné souřadnice, stejné normály a stejné UV koordináty. Zvláště u modelů načítaných z OBJ formátu, kdy spojujeme načtené vertexy do ploch, dochází k duplicitě velice často.

Z hlediska implementace je v případě použití indexování nutné vytvořit nový VBO obsahující pole indexů (stejný postup jako u vytváření jiných VBO). Indexovanou geometrii je pak třeba vykreslovat pomocí funkce *glDrawElements*. Indexování je další možností jak urychlit vykreslování celé scény.

3.2 Shadery

Shader je ve své podstatě program spouštěný na GPU. Slouží k výpočtu vykreslované scény přímo pomocí hardwaru grafické karty, používá tzv. programovatelnou pipeline. Umožňuje vytvá-řet různé druhy grafických efektů (např.: vlnící se vodní hladina), ale také počítat světla, stíny, výpočty teselace, případně měnit geometrii objektů.

Ve spojení s OpenGL používáme GLSL (OpenGL Shading Language) se syntaxí vycházející z jazyka C. Tento jazyk je použit pro tvorbu shader programů obsluhujících jednotlivé programovatelné procesory obsažené v OpenGL, tedy: vertex, tessellation control, tessellation evaluation, geometry, fragment, a compute processors (obr. 1). V této práci je použit vertex procesor a fragment procesor s podporou verzí OpenGL 3.3 a vyšších.



Obrázek 1: OpenGL Render Pipeline

3.3 Shader pool

K práci se shadery je vytvořen v tomto projektu tzv. Shader Pool (viz. také kapitola 4.6). Jedná se o třídu obsahující metody pro správu a hledání shaderů podle jejich ID nebo textového názvu. Instance této třídy je globální v *namespace Shader*, je tedy jednoduše dostupná kdekoliv v enginu a umožňuje tak bez komplikací spouštět načtené shadery. Pro načítání je v této třídě naimplementována metoda *initShader(string jméno vertex shaderu, string jméno fragment shaderu, jméno objektu pro který jsou shadery určeny)*, tato metoda je volána automaticky v případě, že hledáme program který nebyl dříve v aplikaci načten.

Zdrojový kód každého shader programu je uložen v tomto projektu v datech aplikace v adresáři *shaders/*. Adresářová struktura odpovídá objektům ve scéně, pro které je shader určen. Shader je při

startu aplikace pokaždé zkompilován a nalinkován pro konkrétní použitou grafickou kartu, výsledkem je spustitelný program který obsluhuje určitý procesor obsažený v OpenGL.

Pro načtení a kompilaci můžeme použít hned několik knihoven usnadňujících nejen samotné načítání, ale také spouštění, předávání dat apod. Při vývoji jsme testovali *Glew* a *GLee*, tyto knihovny jsou si velice podobné, ale nastávaly komplikace při migraci projektu z Linuxu a kompilátoru gcc do Windows s kompilátorem VC++. Knihovna *GLee* pracující s gcc bez problémů byla ve VC++ nepoužitelná.

3.4 Vertex shader

Program obsluhující vertex procesor se nazývá vertex shader (v adresáři */shaders/*/*.vsh*). Jde o jednotku zpracovávající příchozí vertexy případně jakákoliv přidružená data, který předáme do shaderu jako *uniform*. Tento shader často zpracovává i koordináty textur, umožňuje také modifikovat pozice vertexů a deformovat tak zpracovávaný objekt.

Vertex shader pracuje v čase vždy jen s jedním příchozím vertexem, tento fakt je důležité si uvědomit při jeho návrhu, nenahradí tedy operace, při kterých je třeba znát v jednu chvíli větší část geometrie objektu.

Zdrojový kód 1: Jednoduchý vertex shader

```
#version 420
layout(location = 0) in vec3 vertexPosition_modelspace;

uniform mat4 MVP;

void main()
{
        vec4 v~= vec4(vertexPosition_modelspace,1);

        gl_Position = MVP * v;
}
```

Nejprve uvádíme verzi shaderu (v tomto případě *version 420*, odvíjí se od podporované verze OpenGL). Dále jako *layout* datového typu *vec3* předáváme do shaderu jednotlivé vertexy, pomocí *uniform* datového typu *mat4* předáme model-view projection matici (je vypočtena v enginu jako součin model, view a projection matic). Abychom umístili konkrétní vertex na správné místo ve 3D prostoru musíme jeho relativní pozici vynásobit s MVP maticí (jelikož MVP je maticí 4x4 a vstupní vertex je *vec3*, tedy obsahuje pouze tři hodnoty x,y a z, je třeba jej před násobením přetypovat na *vec4*). Nastavením vestavěné proměnné *gl_Position* předáme upravený vertex dále.

3.5 Fragment shader

Shader obsluhující fragment procesor se nazývá fragment shader (v tomto projektu v adresáři */shaders/*/*.fsh*). Zpracovává jednotlivé fragmenty po rasterizaci, jeho výstupem je barva zpracovaného fragmentu. Také můžeme předávat data pomocí *uniform* stejně jako u vertex shaderu, nejčastěji to bývají textury, případně vlastnosti světla, shadow mapy apod. V tomto projektu je pomocí fragment shaderu řešeno texturování terénu, zpracování světla, stínování, vržené stíny, odrazy vodní hladiny (reflexe), deformace geometrie pod vodní hladinou (refrakce), ale i samotné vlnění vodní hladiny.

Fragment shader nemůže měnit pozici fragmentu, ani není povoleno přistupovat k okolním fragmentům. Data zpracovaná tímto shaderem jsou dále použita k aktualizaci framebufferu, případně textury, záleží jak byl shader spuštěn a kam je v OpenGL směrován jeho výstup (nemusíme tedy kreslit pouze na

obrazovku, ale je možné vytvářet za běhu programu pomocí shaderu např. nové textury a ty dále použít).

Zdrojový kód 2: Jednoduchý fragment shader

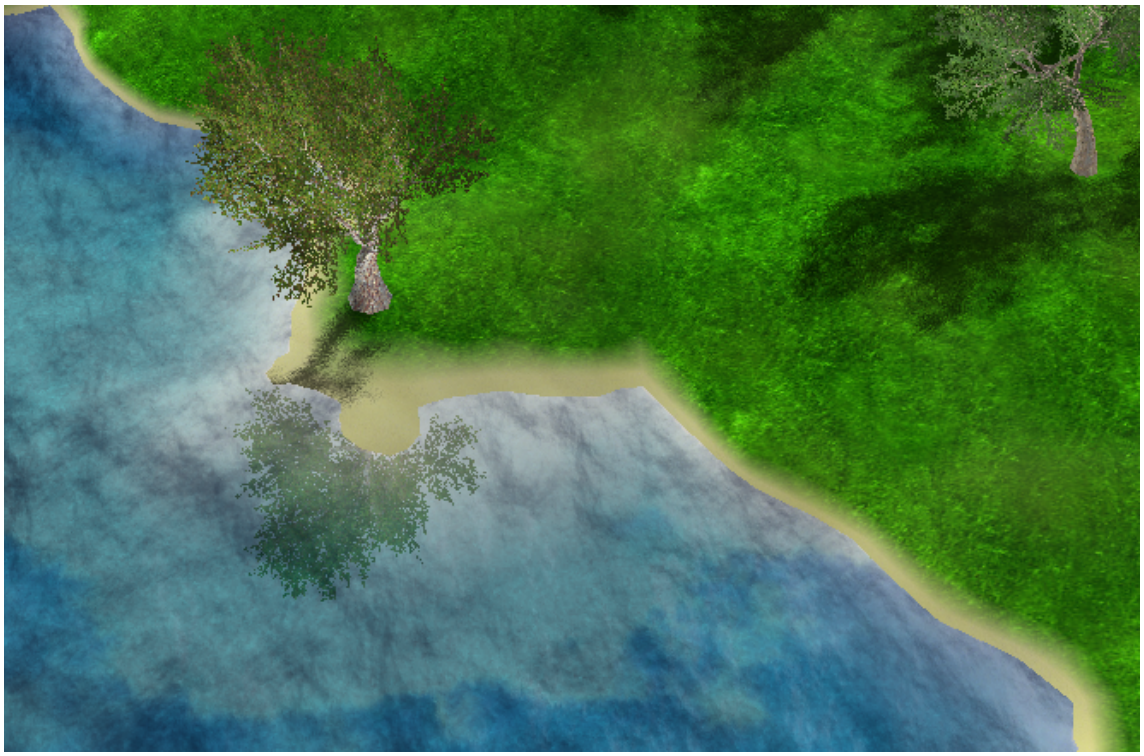
```
#version 420
out vec3 color;

void main()
{
    color = vec3(1,0,0);
}
```

Opět, stejně jako u vertex shaderu, je nutno udat verzi. Dále definujeme výstupní proměnnou *out* datového typu *vec3* s názvem *color*. Ve funkci *main* nastavíme výstupní proměnnou konstantně na *(1,0,0)* (jednotlivé barevné složky RGB). Každý fragment vsupující do tohoto programu bude tedy jednoduše nastaven na červenou barvu.

3.6 Reflexe

Reflexe neboli zrcadlení je nedílnou součástí reálného světa. Existuje několik metod jak tento efekt vytvořit pomocí OpenGL a shaderů. V tomto projektu je použita pro vykreslení realistické vodní hladiny, odrážející plochu terénu, oblohu, ale také objekty na břehu.



Obrázek 2: Ukázka reflexe vodní hladiny

Cílem je vytvořit texturu obsahující zrcadlově otočený obraz objektů, které se nachází nad plochou na kterou bude tato textura použita. Je tedy třeba rozdělit vykreslovaní na dva průchody, v prvním vy-

kreslíme objekty, které se mají na ploše odrážet, do textury. Ve druhém vykreslíme tyto objekty na obrazovku a aplikujeme texturu vzniklou při prvním průchodu.

Funkce pro kreslení reflexe do textury jsou implementovány ve třídě *effects/glReflectoin*, její použití je podobné jako např. *glBegin* a *glEnd* u OpenGL. Nejprve je třeba provést inicializaci při které jako parametr předáváme referenci na vodní plochu. Dále pak pomocí funkce *glReflectionBegin()* přesměrujeme vykreslování scény do textury (rozlišení této textury je stejné jako rozlišení použité pro zobrazení scény, vychází z nastavení enginu v *settings.xml*). Logika této funkce zajistí s použitím *glm::scale(ViewBuffer, glm::vec3(1.0f,-1.0f,1.0f))* přenastavení globální *View* matice tak aby všechny následně kreslené objekty byly zrcadlově otočené podle vodní hladiny.

Všechny objekty vykreslené po volání funkce *glReflectionBegin()* se tedy vykreslují obrácené a do textury, nejsou tedy zatím na scéně viditelné. Pro ukončení kreslení do reflexní textury (všechny objekty, které mají být v odrazu viditelné byly vykresleny) zavoláme funkci *glReflectionEnd()*, jejíž návratovou hodnotou je právě vytvořená reflexní textura. Díky této funkci dojde také k přenastavení vykreslování zpět do framebufferu a navrácení globální *View* matice zpět do stavu před voláním *glReflectionBegin()*.

Zdrojový kód 3: Sekvence tvorby reflexe (první průchod)

```
//init reflection
reflection = new glReflection(water);

reflection->glReflectionBegin();
  if (terVisible) Octree->drawVisible(depthTexture, SHADER_CLIP_UNDER, octreeBox);
  if (skyVisible) sky->draw();  // draw sky
  if (entVisible) entities->drawEntities(SHADER_REFLECTION);
refTexture = reflection->glReflectionEnd();
```

Nyní zbývá vytvořenou texturu (v tomto případě *refTexture*) předat jako parametr funkci pro vykreslení vodní plochy. Dále také vykreslit objekty v bloku *glReflectionBegin()* až *glReflectionEnd()* na obrazovku.

Zdrojový kód 4: Sekvence tvorby reflexe (druhý průchod)

```
//init reflection
if (terVisible) Octree->drawVisible(depthTexture, SHADER_CLIP_UNDER, octreeBox);
if (entVisible) entities->drawEntities(SHADER_REFLECTION);

water->draw(refTexture, refractionTexture);
```

Tato metoda tvorby reflexe ve scéně je vhodná právě pro vodní hladinu a podobné velké plochy, nehodí se pro trojrozměrné objekty u kterých vyžadujeme reflexivitu všech ploch. V takových případech je vhodné použít např. *cube mapu*. Celý objekt zaobalíme do krychle a pro každou stěnu této krychle vytvoříme texturu obsahující části okolní scény, které se mají v daném směru od plochy odrážet. Tyto textury poté namapujeme na původní geometrii objektu.

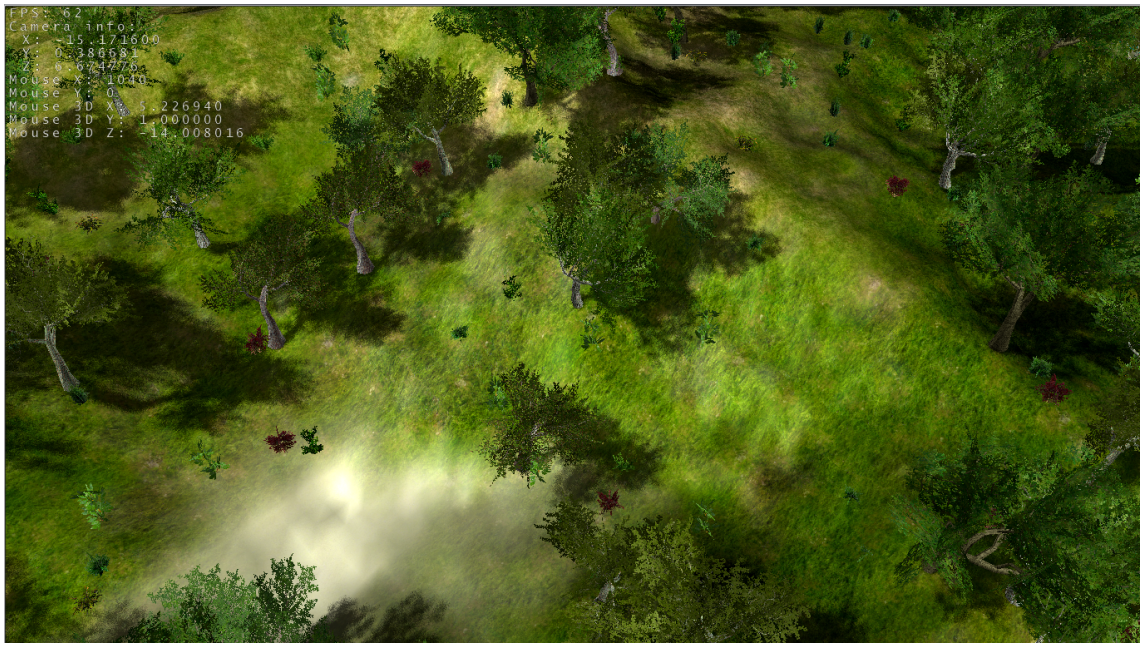
Je patrné, že s použitím reflexe zpomalíme vykreslování scény, jelikož vše (v nejhorším případě) musíme vykreslit dvakrát. Dobré je tedy při implementaci zvážit, které části scény budou v odrazu vodní plochy viditelné a které ne. Pokud například sledujeme scénu z úhlu, ve kterém nenastane situace, kdy by byl viditelný odraz deformací terénu, je zbytečné jej vykreslovat do reflexní textury. Stejně tak je nesmyslné tuto texturu vytvářet v případě, že není vodní plocha ve scéně vůbec viditelná.

Vhodné je umožnit uživateli nastavit, zda se má reflexe zobrazovat, případně mu umožnit nastavit kvalitu jakou bude mít vzniklý odraz (např. změnou rozlišení textury apod.). U komplikovaných a de-

tailních scén se uživatelské nastavení stává naprostou nutností, zpřístupníme tak použití enginu i na méně výkonných počítačích.

3.7 Stíny

Dalším důležitým prvkem reálného světa jsou vržené stíny, opět existuje více metod jak stín vytvořit, jednou z nich jsou tzv. *shadow mapy* (stínové mapy). Tato metoda je v projektu použita.

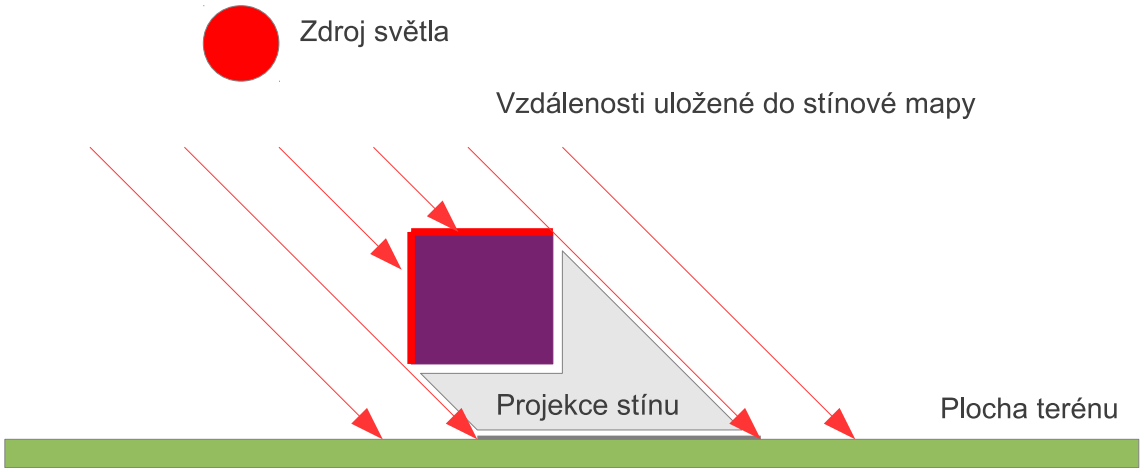


Obrázek 3: Ukázka vržených stínů

Stínová mapa je textura obsahující pouze hloubkovou složku (informace o vzdálenosti objektu od kamery), která se aktualizuje spolu se scénou. Pro její vytvoření je opět nezbytné rozdělit vykreslování na dva průchody. V prvním vytvoříme stínovou mapu, jednoduše vykreslením scény z pozice světelného zdroje a ve druhém ji aplikujeme na plochu kam má stín dopadat, tedy zkoumáme každý fragment, zda se nachází ve stínu (je dál v prostoru než fragment uložený ve stínové mapě). Pokud zjistíme, že je ve stínu, změníme jeho barvu (např. vynásobením všech barevných složek konstantou).

Metody obsluhující tvorbu stínů jsou naimplementovány ve třídě *effects/glShadow*. Jejich použití je stejné jako v případě reflexe, po inicializaci je před blokem, kde bude vytvořena stínová mapa volána funkce *glShadowBegin()*. V této funkci je přesměrováno vykreslování scény do textury (její rozlišení je nastavitelné, testováno bylo 2048x2048), uchováváme pouze hloubku scény, barevné komponenty jsou v tomto případě nepoužité a je zbytečné je ukládat. Při vykreslování do stínové mapy je použita jiná projekční matice než pro zobrazenou scénu, její nastavení se liší podle použitého světla. Pro všesměrový bodový zdroj světla (chová se jako Slunce, osvětluje rovnoměrně celou plochu terénu) je použita ortografická projekce, pro směrový zdroj světla se používá perspektivní projekce. V tomto projektu je světlo ozařující rovnoměrně celou scénu, je tedy použita ortografická projekce. Po nastavení projekční matice dojde ke změně *view* matice tak aby bylo na scénu nahlíženo z pohledu světelného zdroje.

Všechny objekty vykreslené po volání *glShadowBegin()* se přidávají do stínové mapy jako objekty vrhající stín. Jelikož nás zajímá pouze hloubková informace, je vhodné použít pro jejich vykreslení jednoduché shadery, sloužící pouze pro zobrazení geometrie bez textur, materiálů apod. V opačném případě



Obrázek 4: Schématické znázornění vytváření stínové mapy

dojde ke zbytečnému zpomalování celého vykreslovacího procesu. K ukončení bloku tvorby stínové mapy slouží metoda *glShadowEnd()*, její návratovou hodnotou je stínová mapa. Dojde také k nastavení původní projekční matice a *view* matice.

Zdrojový kód 5: Sekvence tvorby stínů (první průchod)

```
shadows->glShadowBegin();
  if (terVisible) Octree->drawVisible(0, SHADER_SHADOW, octreeBox);
  if (entVisible) entities->drawEntities(SHADER_SHADOW);
GLuint depthTexture = shadows->glShadowEnd();
```

Stínová mapa *depthTexture* v tomto případě obsahuje stíny všech statických, viditelných objektů na scéně (*Octree-drawVisible*) a pohyblivých entit (*entities-drawEntities*). Následuje sekvence, kdy předáme *depthTexture* funkci pro vykreslení plochy terénu.

Zdrojový kód 6: Sekvence tvorby stínů (druhý průchod)

```
if (terVisible) Octree->drawVisible(depthTexture, SHADER_NORMAL, octreeBox);
```

Samotné zpracování stínové mapy a zobrazení stínů obsluhuje shader program terénu. Posledním krokem je zjistit zda právě zpracovávaný fragment plochy terénu je dále v prostoru než fragment uložený ve stínově mapě, pokud ano, leží ve stínu. Pro výpočet koordinátů stínové mapy je třeba převést ve vertex shaderu vstupní vertexy do prostoru ve kterém byla vytvořena, tj. vynásobit *model-view-projection* maticí (dále MVP), která byla použita při vytváření stínové mapy (dále *depthMVP*). Nastává zde však problém, že pouhým vynásobením vertexů s *depthMVP* získáme homogenní koordináty (záporné i kladné), ale pro mapování textur jsou třeba koordináty od 0 po 1. Tento problém řeší vynásobení *depthMVP* s *Bias* maticí (viz. vzorec (2)), tedy vynásobení každého prvku konstantou 0,5 (diagonála) a posun o 0,5 (poslední řádek matice).

$$Bias = \begin{pmatrix} 0.5 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.0 \\ 0.5 & 0.5 & 0.5 & 1.0 \end{pmatrix} \tag{1}$$

$$depthBiasMVP = Bias \cdot depthMVP \tag{2}$$

Jelikož matice *depthBiasMVP* musí být dostupná jak z vertex shaderu, tak z fragment shaderu, je její výpočet proveden na CPU a je předána jako *uniform* do obou shader programů.

Vertex shader:

Zdrojový kód 7: Použití stínové mapy vertex shader

```
ShadowCoord = depthBiasMVP * vec4(vertexPosition_modelspace,1);
```

Kde *ShadowCoord* jsou koordináty stínové mapy a *vertexPosition_modelspace* jsou pozice jednotlivých vertexů zpracovávaného objektu. Koordináty stínové mapy jsou dále předány do fragment shaderu.

Fragment shader:

Zdrojový kód 8: Použití stínové mapy fragment shader

```
float visibility = 1.0;

if ( texture( shadowMap, ShadowCoord.xy ).z  <  ShadowCoord.z)
    visibility = 0.5;
```

Kde *visibility* je hodnota kterou je násobena výstupní barva fragmentu, *shadowMap* je stínová mapa předána jako *uniform*, *texture(shadowMap, ShadowCoord.xy).z* je vzdálenost mezi světlem a fragmentem na dané pozici ve stínové mapě, *ShadowCoord.z* je vzdálenost mezi světlem a aktuálním fragmentem terénu. Pokud je aktuální fragment dále než fragment ve stínové mapě, tak se nachází ve stínu a jeho barva je změněna.

Podobně jako je tomu u reflexe, je i u stínů vhodné umožnit uživateli nastavit jejich kvalitu a zda se mají vůbec zobrazovat. Jelikož je třeba rozdělit vykreslování na více průchodů, ve kterých je třeba geometrii, u které vyžadujeme aby vrhala stín, vykreslit jednou při tvorbě stínové mapy a podruhé při jejím normálním zobrazení, dochází k dalšímu zatěžování GPU a CPU.

Další metody tvorby vržených stínů:

Projection shadows: objekt vrhající stín promítneme z pohledu od světelného zdroje na plochu kam stín dopadá. Tato metoda je nevhodná v případech, kdy chceme aby stín dopadal na komplikovanější objekty než je pouze rovná plocha.

Shadow Volumes: při použití této metody vytváříme v prostoru nový objekt *shadow volume* reprezentující stín a zkoumáme zda nedochází ke kolizi s jiným objektem. Pokud taková kolize nastane je část nebo i celý objekt ve stínu. Tento algoritmus může být jednoduše rozšířen pro více světelných zdrojů ve

scéně, na druhou stranu nastává problém, pokud je objekt vrhající stín komplikovaný, může být obtížné vytvořit korektní *shadow volume*.

3.8 Vykreslování scény

Hlavní funkce pro vykreslení celé scény je naimplementována ve třídě *Map*, funkce *drawAll*. Vykresluje vše na mapě, vykreslení GUI apod. je řešeno odděleně. Jak bylo popsáno výše, celá scéna je složena z několika částí a nevykresluje se pouze jednou, ale v závislosti na použitých efektech. Ve finálním ukázkovém demu vypadá následovně:

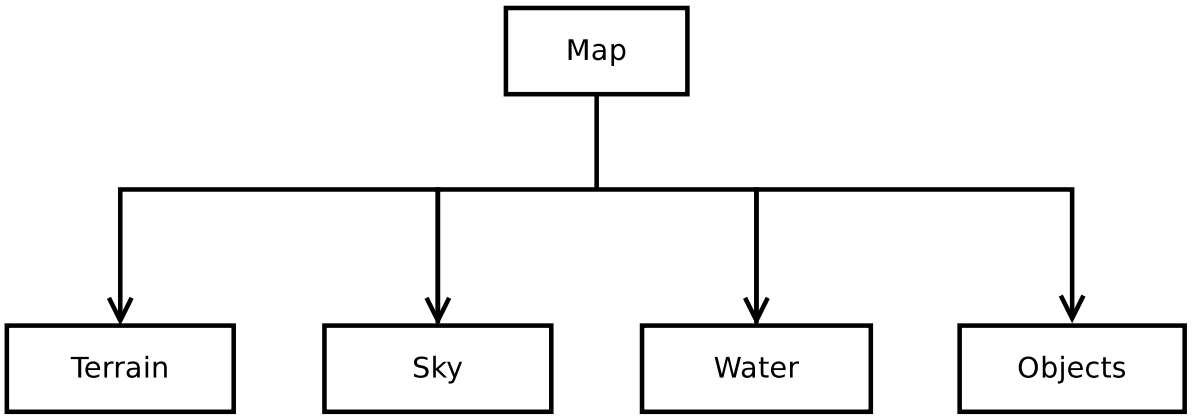
- Tvorba stínů - (viz. kapitola 3) zde je vytvořena stínová mapa, vykreslovány jsou pouze entity. Žádné jiné objekty v ukázkovém demu nevrhají stín.
- Refrakce - do textury se vykresluje pouze terén a to jen jeho část pod vodní hladinou. Vzniklá textura je pak aplikována na vodní plochu, pomocí fragment shaderu je deformována v závislosti na vlnění vodní hladiny.
- Reflexe - (viz. kapitola 2) vykreslena je obloha a entity. Jelikož je reflexní textura použitá na vodní plochu, dochází také ke kontrole zda je voda aktuálně viditelná, pokud ne, je celý tento blok přeskočen.
- Terén a statické entity - je vykreslována pouze část terénu nad vodní hladinou, od tohoto bloku se vykresluje už pouze na obrazovku.
- Pohyblivé entity - vykreslení všech pohyblivých entit na mapě.
- Ostatní - vykreslení kurzoru (pro označování jednotek), případně vykreslení bounding boxů entity (pouze pokud je uživatelsky povoleno v konzoli enginu)

4 Architektura

4.1 Struktura enginu

Námi vyvíjený game engine je dělený do několika vrstev starajících se o konkrétní oblasti ve hře, tyto vrstvy mezi sebou komunikují, předávají si reference, data a tvoří tak jeden celek. Na toto dělení se můžeme dívat z několika pohledů. V rámci urychlení a nezávislosti aktualizací stavu jednotek a renderu je engine rozdělený do dvou vláken. Hlavní vlákno ze kterého je aplikace spouštěna obstarává inicializaci dat (načítání textur, modelů, kompilace shaderů, nastavení mapy a její vygenerování apod.) a vykreslování scény, druhé vlákno spuštěné po dokončení inicializace obstarává správu událostí (uživatelských i programově vytvořených), aktualizaci stavů jednotek (změna pozice při pohybu, reakce na kolizi, hledání cesty apod.).

Dale můžeme dělit engine podle jednotlivých datových struktur a objektů starajících se o určitou část vytvořené scény. Jedná se o stromovou strukturu, kde kořenem je třída Map (obr. 5).



Obrázek 5: Složení mapy

Dále se tedy zaměříme na jednotlivé části mapy.

4.2 Terrain

Třída reprezentující plochu terénu na mapě, obsahující metody pro generování deformací pomocí diamod-square algoritmu (viz. kapitola 5), vykreslení tohoto terénu, metody pro analýzu vygenerovaných deformací apod. Terén je složen z jednotlivých vertexů tvořících síť, vzdálenosti mezi vertexy určuje nastavitelná velikost kroku.

Pro dosažení realistického vzhledu terénu, je použito multitexturování, řízené výškovou hodnotou (y souřadnicí) daného fragmentu. Míchání textur je realizováno pomocí fragment shaderu terénu, kde se mění poměry viditelnosti mezi jednotlivými vrstvami textur. Do shaderu jsou předávy textury pro oblast pod vodní hladinou, přechod z vody na travnatou plochu, přechod z travnaté plochy na skálu a skálu. Pro odstranění dojmu opakující se textury (zejména na velkých plocháh ve stejné výšce) je celý povrch mapy pokryt částečně průhlednou texturou obsahující deformace terénu apod.

Tato třída obsahuje také metody pro vytvoření VBO objektů a jejich nabindování do GPU, můžeme zvolit určitou část terénu a vytvořit pro ni VBO (použito při dělení scény viz. kapitola 7.4)

4.3 Sky

Třída reprezentující oblohu ve scéně. Jedná se o jednoduchou texturovanou plochu, která je vykreslována pouze při tvorbě reflexní textury (viz. kapitola 3.6) pro vodu. Koordináty textury se v čase mění

pro vytvoření dynamické oblohy.

4.4 Water

Třída reprezentující vodu. Probíhá u ní podobné dělení jako u terénu (viz. kapitola 5). Textura je složená z obrazu části terénu pod hladinou (probíhá zde refrakce), odrazu objektů nad hladinou (reflexe) a z normálové textury vody. Efekt vlnící se hladiny je realizován pomocí shaderů, přepočítáváním normálové mapy. Voda je zaznamenána jako neprůchozí oblast a žádné pohyblivé jednotky přes ni nemohou projít, ani v její oblasti být vytvořeny.

4.5 Objects

Tato třída reprezentuje všechny objekty (entity) na mapě. Kromě metod pro vkládaní nových objektů jsou zde pointery na tyto objekty také uloženy.

Konkrétní jednotky se svými konkrétními vlastnostmi jsou vytvořeny pomocí dědičnosti z nadřazených tříd, kde je nejvýše třída entity obsahující metody společné pro všechny objekty na mapě, jako je vykreslování, metoda control (volaná pouze z vlákna pro obsluhu událostí, obstarávající update entit), metody pro nastavení modelů entity apod. Z této hlavní třídy dědí *StaticEntity* a *DynamicEntity*, což je rozlišení objektů na mapě na pohyblivé a nepohyblivé. U pohyblivých entit najdeme rozšíření o metody pro zadání pohybu z aktuální pozice na pozici novou (s využitím pathfinderu pro nalezení cesty v terénu). Strom dědičnosti dále pokračuje a konkretizuje vlastnosti entit.

Nespornou výhodou takovéto struktury je zjednodušení a sjednocení přístupu k jednotlivým objektům, můžeme totiž zobecnit všechny jednotky na mapě jako datový typ *Entity* a jednoduše volat například metody pro vykreslování nebo jakékoliv společné metody. Pro identifikaci konkrétního datového typu obsahuje každý objekt identifikátor *int type* udávající o jaký konkrétní typ entity se jedná, ale také *int parentType* udávající rodičovskou třídu výše ve stromu. Pokud tedy zjistíme o jaký typ se jedná, není už problém přistupovat k danému objektu např. jako k budově, stromu nebo zvěři a používat jejich specifické metody.

Z hlediska implementace potřebujeme s postupem času spíše upravovat a rozšiřovat strukturu o nové typy jednotek (např. dřevorubec, lovec, stavitel, apod.), dědičnost nám umožní používat již dříve naprogramované metody a rozšířit o metody nové zaměřené na určitou problematiku. Nemusíme tedy nic kopírovat, kód se stává přehlednějším a případné chyby se lépe hledají.

Nabízí se také možnost konkrétní listové třídy implementovat, alespoň z části, pomocí skriptů, získáme tak možnost měnit vlastnosti jednotek aniž bychom museli přepisovat samotný engine a znovu kompilovat.

Organizace objektů:

Jak bylo řečeno dříve, v této třídě jsou uloženy i pointery na jednotlivé objekty, a to v kontejneru *vector*. Je tedy možné přistupovat k jednotlivým objektům nejen pomocí jejich adresy, ale také pomocí id ve vectoru. Tento kontejner je vhodný pro rychlé procházení velkého počtu záznamů (s použitím id prvku, ne pomocí iterátoru), ale také pro dynamické přidávání nových prvků.

Problém však nastává v případě, že je potřeba uložený objekt odstranit (jednotka byla zničena, vymazána apod.). Pokud se nachází na konci vectoru je možné ji jednoduše odstranit pomocí metody *pop_back()*, to ovšem neřeší situaci, kdy zanikne objekt uložený jinde než na konci. Jelikož je třeba zachovat všechny ostatní prvky na jejich původním id (z důvodu přístupu k jednotlivým objektům), není možné přemístit prvek, který chceme odstranit na konec fronty a poté jej smazat (došlo by k přečíslování ostatních prvků a tedy ke zmatku v uložených datech). Mazání objektů je tedy realizováno s pomocí

dalšího vectoru (*freeID*) obsahujícího id objektů, které jsme odstranili. Entita, u které vyžadujeme smazání, se pouze přepne do stavu smazaná, čímž zastaví funkci vykreslování a funkci *control*, ale stále existuje v paměti. Její id se přidá do *freeID* vectoru. V případě, že je potřeba přidat do scény nový objekt, dojde nejprve ke kontrole *freeID*, jestli obsahuje nějaký záznam o volném místě ve vectoru s objekty. Pokud ano, je použito, v opačném případě je vector s objekty rozšířen o nový záznam.

4.6 Přístup k datům

Jedním z prvotních problémů, které bylo třeba při implementaci vyřešit byl přístup k datům. Jak efektivně načítat data ze souborů a v RAM paměti držet jen ta jedinečná (např. několik entit co jsou právě na mapě má stejné modely, používá stejné shader programy a textury). Je tedy nesmyslné načítat, při každém vytvoření nového objektu, data znovu a zbytečně tak zahlcovat RAM paměť a zpomalovat běh enginu. Účinným řešením tohoto problému je použití tzv. poolů, držících název načteného souboru a adresu načtených dat v paměti. Tato struktura je použita pro načítání textur, shaderů a modelů.

Sekvence načítání shaderů: nejprve proběhne inicializace shader poolu (při startu enginu), kde můžeme, ale nemusíme načíst shader programy o kterých jistě víme že budou dále v programu použity (pomocí metody *loadDir* můžeme načíst všechny shader programy v adresáři). Pokud se dále dostaneme k místu, kde je třeba shader použít, můžeme se pokusit zjistit jeho ID pomocí metody *getProgramID*, jejím parametrem je název shaderu který hledáme. Vnitřní logika shader poolu zjistí zda tento shader program byl už načten, pokud nebyl, načte jej a vrátí zpět jeho ID. Jelikož hledání v poolu je časově náročné (dochází k cyklické komparaci stringů uložených názvů a hledaného názvu) je vhodné načíst shader při inicializaci (např. konkrétní entity) a držet si jeho ID. Při testování případu, kdy se pokaždé hledal shader podle názvu bylo zpomalení hlavního vlákna neúnosné.

Načítání ostatních dat (textur, grafiky pro GUI a modelů) probíhá stejně, jednou z nevýhod takového přístupu je rozlišování dat pouze podle názvu souborů. Může nastat situace, kdy načítáme jiná data se stejnými názvy, ale z jiných adresářů (např.: */textures/tex.png* a */textures/another/tex.png*). V takovém případě by došlo k načtení pouze prvního souboru, druhý už by byl ignorován. Tomuto problému se dá předejít například uložením nejen názvu souboru, ale i cesty.

5 Generování prostředí

5.1 Výhody a nevýhody

Z hlediska použití v RTS strategii je generování prostředí žádoucí, často potřebujeme vytvořit zcela náhodné herní prostředí a vyhnout se tak stereotypu, který by mohl nastat po delší době hraní. Nevýhodou takového prostředí je jeho obtížná kombinace s případným příběhem, dějem konkrétní mise nebo celé kampaně hry. V takovém případě je vhodnější vytvořit pro engine editor mapy, kde provážeme vlastnosti terénu s požadavky hry a vytvoříme tak komplexní celek. Případně se nabízí možnost tyto dvě metody zkombinovat a umožnit uživateli výběr, zda hrát náhodnou misi nebo konkrétní připravenou kampaň (příkladem může být série her Age Of Empires).

5.2 Definice

Organizace dat

Terén je reprezentován v kódu dvourozměrným polem *float* hodnot, kde prvním indexem tohoto pole je souřadnice x, druhým z souřadnice a uloženou hodnotou na těchto souřadnicích je y (výšková hodnota daného bodu). Takto je vytvořena síť následně propojená do trojúhelníkových ploch při vykreslování.

Eroze

Efekt eroze je velice obtížné popsat matematicky, termín eroze zahrnuje velké množství možných deformací, na různých typech terénu a závisí také na klimatu ve kterém tyto deformace vznikají [3]. Existuje však několik možností jak tyto eroze simulovat a alespoň částečně tak přiblížit vzhled vygenerované plochy realitě. Jednou z metod jak vygenerovat základní deformaci plochy terénu je použití *Diamod-square algoritmu*, který je popsán v následujícím textu. Na vzniklou plochu je možné dále aplikovat různorodé erozní algoritmy a pomocí syntézy jejich výstupů vytvořit realistickou scénu.

5.3 Diamod-square algoritmus

Diamod-square algoritmus neboli metoda posunu středního bodu je v tomto enginu použita pro vygenerování náhodné plochy terénu. Pracuje rekurzivně, jejím vstupem je dvourozměrné pole bodů, výstupem je toto pole se změněnou výškovou hodnotou každého bodu. Jedná se o fraktálový algoritmus velice se přibližující svým vzhledem růžovému šumu.

Popis funkce algoritmu

Jednotlivé kroky jsou popsány v návaznosti na obr. 6.

a) V tomto kroku jsou vybrány rohové body (dále *A, B, C, D*) celého terénu a jejich výškové hodnoty jsou nastaveny náhodně, v požadovaném rozsahu (dále *R*).

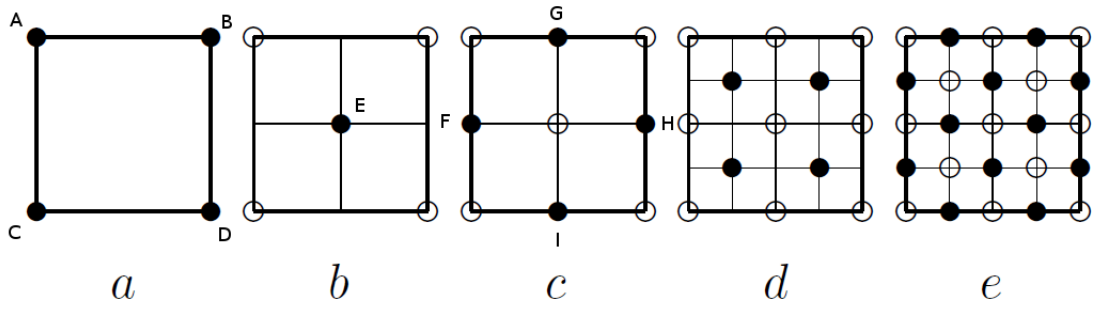
b) Je určen střední bod *E* a jeho výšková hodnota je nastavena podle vztahu (3).

$$E = \frac{(A + B + C + D)}{4} + random(R) \tag{3}$$

c) Zde jsou vybrány body ležící ve středu stran (dále *F, G, H, I*). Jejich výškové hodnoty jsou nastaveny podle vztahu (4).

$$F = \frac{(A+C)}{2} + random(R), G = \frac{(A+B)}{2} + random(R), ... \tag{4}$$

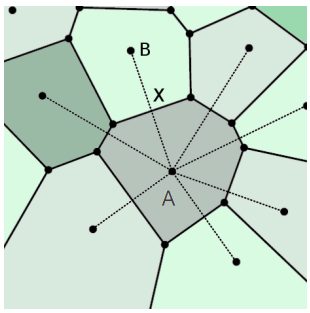
d) Nyní je původní čtverec rozdělen na čtyři menší a nachází se v místě, kde vzniká dříve zmíněná rekurze. Stejný postup je aplikován na nové menší čtverce. Úroveň zanoření závisí na počtu bodů, na které je plocha dělena. Rozsah *R* se s každým novým zanořením snižuje. Rekurze je ukončena po nastavení všech bodů vstupního pole.



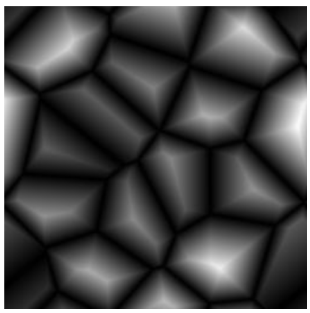
Obrázek 6: Jednotlivé průchody Diamod-square algoritmu (převzato z [3])

5.4 Voroného diagram

Terén vygenerovaný pomocí diamond-square je z velké části homogenní a izotropní, tedy nemá vlastnosti reálného terénu. Postupem času zjistíme, že takto vygenerované plochy jsou si navzájem velice podobné. Jednou z metod jak se vyhnout tomuto stereotypu je aplikace voroného diagramů, mimo jiné používaných také pro vytváření procedurálních textur a dalších efektů. Pro vytvoření výškové mapy pomocí těchto diagramů, jsou umístěny na vstupní plochu náhodně *n* referenční body (obr. 7 např. body *A, B*). Poté je určen střed vzniklé přímky AB bod *X*, ten určí hranici buňky. Stejný postup je aplikován na všechny referenční body a dojde tak k rozdělení celé plochy. Výškové hodnoty samotných bodů terénu jsou vypočítány podle vztahu (5).



Obrázek 7: Dělení do buněk



Obrázek 8: Výsledná výšková mapa

$$h = \sum_{i=1}^{n} c_i \cdot d_i \tag{5}$$

Kde d_i je vzdálenost mezi bodem terénu, u kterého je zjišťována výšková hodnota a referenčními body, c_n je určený koeficient (ovlivňuje vlastnosti vygenerované plochy) a *h* je výška daného bodu [3].

5.5 Kombinace obou algoritmů

Samotné voroného diagramy nemohou být použity pro dosažení realistického vzhledu plochy terénu. Nejlepších výsledků dosáhneme kombinací obou těchto algoritmů, takto vytvoříme tzv. kombinovanou výškovou mapu terénu, přibližující se svým vzhledem reálnému terénu. Naskýtá se také možnost více ovlivnit výsledné vlastnosti a vzhled terénu, čímž snížíme pocit stereotypu herního prostředí. Dobrým příkladem použití procedurálního terénu je mimo jiné hra Tribal Trouble, kde jsou výše popsané metody použity pro vytvoření rozmanitých a velice realistických ostrovů.

5.6 Umístění entit na mapě

Pro procedurální vytvoření mapy nestačí pouze vygenerovat plochu terénu, je třeba na ni umístit také entity dotvářející herní prostředí. V tomto enginu je při volání funkce *generateMap* umístěné ve třídě *Map*, spuštěno, po vygenerování terénu, také náhodné umisťování entit na mapě. Takto je po celé herní ploše rozmístěna vegetace. Je nalezena náhodná, neobsazená pozice, nad vodní hladinou, na kterou je následovně umístěna entita (např. zvěř, strom, keř apod.). Umístěný objekt je také náhodně zarotován a jeho věk *age* (parametr entit) je také náhodně nastaven. Věk zvláště u stromů a keřů určuje jejich velikost, s časem se zvětšuje až po dosažení maximální hodnoty.

Takové řešení není příliš vhodné pokud chceme vytvořit skutečně realistické prostředí (např. les). Je použito pouze pro demonstraci v ukázkové aplikaci, abychom nemuseli vkládat vše ručně. Nabízí se řešení, kdy např. po vytvoření jednoho stromu umístíme další v jeho omezeném okolí. Počet stromů v oblasti i oblast samotná mohou být nastaveny náhodně, čímž dojde k vzniku různorodých skupin entit.

6 Kolize mezi entitami

I nejjednodušší 2D hra potřebuje detekci kolizí. Jedná se o jednu ze základních interakcí mezi objekty a proto se bez ní nelze obejít. Kolize je matematickou kalkulací průniku objektů jako jsou body, přímky, plochy nebo polygony ať už v dvou nebo trojrozměrném prostou. S detekcí kolizí souvisí také samotná logika hry, tedy jak daný objekt bude reagovat při zjištění kolize s jiným, ale také s fyzikou těchto objektů.

6.1 Bounding box

Z hlediska náročnosti, není vhodné detekovat kolize hledáním průniku celé geometrie objektů. Zvláště u her typu RTS není třeba příliš přesná a detailní detekce, navíc vzhledem k často velkému počtu objektů by byl tento způsob příliš časově náročný. Jednou z metod jak se vyhnout zkoumaní komplikovaných objektů je zaobalit celou jejich geometrii do tzv. bounding boxů. Tedy hledáním nejmenšího možného kvádru, který obsáhne všechny vertexy modelu. Bounding box může být vytvářen procedurálně za běhu programu nebo je možné jej načíst spolu s modelem. Každá z těchto možností má své výhody a nevýhody, ne vždy totiž potřebujeme nalézt box který by obsáhl celou geometrii, můžou nastat situace, kdy můžeme vyžadovat určitý offset apod. U předefinovaných boxů je tedy možné vytvořit zcela nezávislou kolizní oblast, takovou, jakou u daného objektu požadujeme. Na druhou stranu nastává problém např. při změně velikosti modelu, kdy musíme změnit ručně i boundimg box.

V tomto enginu je použita druhá metoda, tedy každý objekt má svůj model a svůj bounding box, určující jeho kolizní oblast, předem definovaný. Model i box jsou načítány najednou pomocí třídy *modelPool*. Samotný box je objektem třídy *Cube* obsahující metody pro detekci kolizí (detekce kolize dvou boxů, box-bod apod.).

Typy bounding boxů

- AABB (Axis-Aligned Bounding Box) box souběžný s osami souřadnicového systému
- OBB (Oriented Bounding Box) box zarotovaný vůči osám

2D kolize ve 3D prostoru

Při návrhu kolizního systému je dobré zhodnotit jaké situace mohou vůbec nastat. U tohoto enginu jsou všechny objekty umístěny na ploše terénu, tedy i jejich bounding boxy. Proto je možné zjednodušit detekci kolizí promítnutím celé mapy a objektů na ní ležících do 2D prostoru. Terén je orientován svou plochou na osy x a z, osa y určuje výšku terénu v určitém bodě. Při hledání kolizí tedy můžeme kontrolovat pouze průniky základen bounding boxů jednotlivých objektů. Rozměr y tedy můžeme, při detekci kolizí mezi objekty, ignorovat. Takové zjednodušení však nelze použít vždy, například při hledání objektu na který bylo kliknuto myší je nutné kontrolovat kolizi s celým boxem.

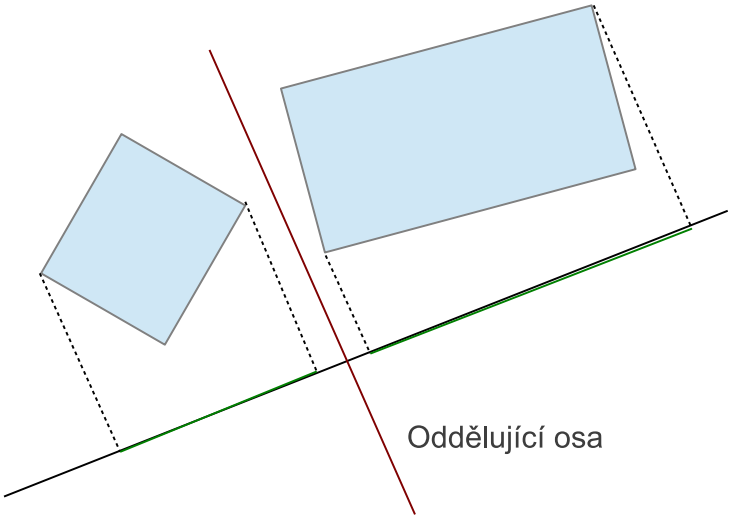
6.2 Detekce kolizí

Kolize dvou AABB

Kolize dvou bounding boxů souběžných s osami je naimplementována ve třídě *Cube* ve funkci *cubeInCube*, jako parametr předáváme box u kterého zkoumáme zda je v kolizi. Návratovou hodnotou je datový typ *bool*, a nabývá hodnoty *true* v případě že dochází ke kolizi. Logika funkce porovnává průniky obou boxů ve všech rozměrech, pokud k žádnému nedojde, boxy nekolidují.

Kolize dvou OBB

Detekce kolize dvou zarotovaných boxů je mnohem komplikovanější. Nestačí pouze triviální kontrola přesahů. Pokud se snažíme nalézt takovouto kolizi v reálném světě, musíme nahlédnout na oba objekty z více stran, nekolidují v případě, že z jakéhokoliv pohledu mezi nimi vidíme ”mezeru”. Stejný princip je použit i v tomto případě.



Obrázek 9: Detekce kolize dvou OBB

Jedná se o případ použití *Věty o oddělujících osách* která zní pro tento případ následovně: Jestliže se dva OBB boxy nedotýkají, pak existuje oddělující osa $\vec{L} = \vec{V} \times \vec{W}$, kde $\vec{V}$ a $\vec{W}$ jsou ruzné vektory vybrané z šesti základních os dvou bounding-boxů. Tedy zkoumáme patnáct oddělujících os: tři základní osy obou OBB a devět os vzniklých vektorovým součinem mezi základními osami zkoumaných OBB. Pokud je jedna z těchto os také oddělující osou, kolize mezi objekty neexistuje. Pokud vyjdeme z předpokladu, že v tomto enginu nenastává situace, kdy by objekt (i s bounding boxem) neležel na ploše terénu, je možné zjednodušit detekci kolize na hledání oddělující osy dvou zarotovaných obdélníků (počítat pouze se základnami OBB). Počet kontrolovaných os se tedy omezí na dvě základní osy obou obdélníků a jejich vektorové součiny [9, 10].

7 Culling

7.1 Princip

Při tvorbě grafického subsystému se často setkáváme s problémem jak vykreslit detailní a realistickou scénu v co nejkratším čase, nezahlcovat grafickou kartu zbytečnými daty, která ve výsledku nejsou na obrazovce v daný okamžik viditelná. Zvláště u her typu RTS nastávají až velice často situace, kdy máme na herní ploše stovky objektů (celá města, velké skupiny pohybujících se jednotek, vegetaci, zvěř apod.) kdybychom vykreslovali vše, bez ohledu na jejich viditelnost, pravděpodobně by byl náš engine pomalý a nepoužitelný.

Soubor metod na vyloučení neviditelných částí scény se nazývá culling. Těchto metod existuje velké množství a některé z nich budou posány v následujícím textu, zejména ty, které jsou vhodné pro hry typu RTS.

7.2 Frustum culling

Frustum culling identifikuje objekty ležící mimo oblast, kterou zobrauje kamera. Tato oblast je tvořená komolým jehlanem, opisujícím vlastnosti perspektivy použité v dané scéně. U každého objektu je tedy zkoumána, při změně jeho pozice, nebo při pohybu kamery, kolize jeho bounding boxu a komolého jehlanu viditelné oblasti. Pokud tato kolize není detekována, je objekt označen jako právě neviditelný a není volána funkce pro jeho vykreslení [5]. Vyřazeny z vykreslovacího procesu jsou tedy také objekty ležící příliš blízko kamery a mimo dohled kamery.

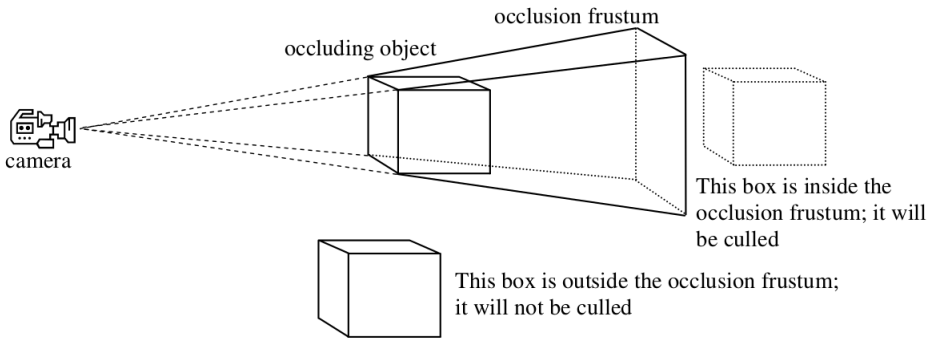
Implementace

Tato metoda cullingu je v našem enginu přímo použita pouze u pohybujících se objektů (jako je zvěř, jednotky apod.). Její logika je částečně obsažena ve statické globální třídě *World*, kde je při každé změně pozice kamery přepočítána viditelná oblast. Každý pohyblivý objekt pak při aktualizaci své pozice zjišťuje, zda je stále viditelný (kontrolou kolize svého bounding boxu a viditelné oblasti), podle této informace je následovně nastavena *bool* proměnná, určující zda bude entita v následujícím průchodu vykreslena.

7.3 Occlusion culling

Occlusion culling je další metodou jak zamezit zbytečnému zpracovávání neviditelných objektů. Dochází k vyloučení objektů, v aktuálním pohledu, skrytých za jinými objekty. Dojde, podobně jako je tomu u frustum cullingu, k vytvoření komolého jehlanu (dále *occlusion frustum*) kopírujícího vlastnosti aktuální perspektivy a tzv. occluding objektu (viz. obr. 10). Occlusion frustum tedy zahrnuje celý prostor, který nemůže být při dané pozici kamery a occluding objektu viditelný. Všechny objekty, které jsou celé v prostoru occlusion frustum jsou tedy neviditelné a mohou být vyřazeny z vykreslování [5].

V závislosti na typu grafického systému a typu zobrazených objektů může frustum culling, occlusion culling, případně kombinace obou značně eliminovat množství zpracovaných objektů. Máme tak možnost jak redukovat počet zpracovávaných vertexů, textur atd. a zrychlit tak celý běh systému. S jejich použitím můžeme dosáhnout rychlého zobrazování komplikovanějších scén. V našem případě je použití occlusion cullingu nevhodné, vzhledem ke statickému pohledu (pod neměnným úhlem) na scénu nenastává situace, kdy některý objekt plně překrývá jiný, příliš často. Vyhledávání objektů, které by měly být odstraněny ze zpracování by zbytečně zpomalovalo tento engine. Avšak nelze to tvrdit o všech hrách typu RTS, použití těchto algoritmů je třeba zvážit.

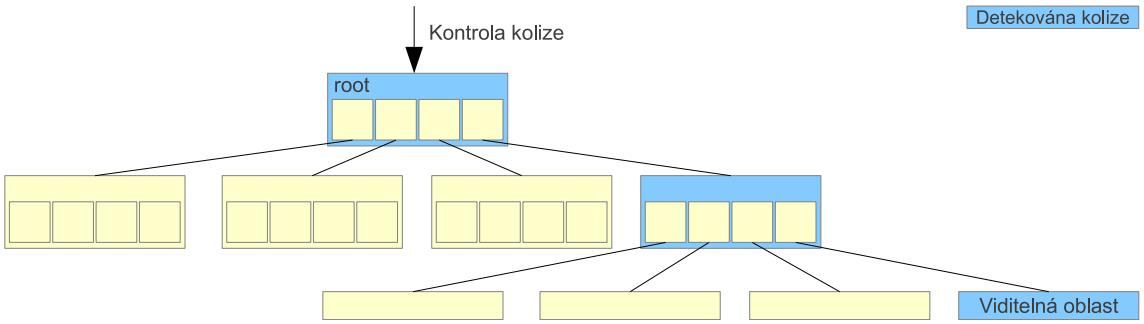


Obrázek 10: Occlusion culling (převzato z [11])

7.4 Quadtree

Plocha terénu je, stejně jako ostatní geometrie, složena z velkého množství vertexů. Je tedy nereálné zkoumat u každého zda je v danou chvíli viditelný. Podobně jako tomu je u pohyblivých objektů (s použitím bounding boxů) je třeba rozdělit terén na oblasti zaobalené do boxů a řešit viditelnost těchto boxů. K tomuto dělení je použit v tomto enginu quadtree. Jde o datovou stromovou strukturou, kde je každý uzel rozdělen vždy na čtyři další (kromě listových uzlů). Uzlem je myšlen čtverec (v našem případě box) obsahující určitou část scény.

Samotné dělení probíhá rekurzivně, nejprve je určen kořen stromu (obaluje celou geometrii terénu). Dále jsou vytvořeny čtyři nové uzly obalující odpovídající část geometrie nadřazeného uzlu (poprvé kořenový uzel). Toto dělení se opakuje pro každý nově vzniklý uzel, dokud nedosáhne úroveň zanoření své maximální hodnoty (předem určena). Konečné listové uzly pak obsahují samotná data (každý pro svou oblast) terénu a obstarávají jejich vykreslování. Takovéto uspořádání, do stromové struktury, umožňuje jednoduše a rychle vyřazovat neviditelné oblasti [5]. Pokud budeme stromem procházet od kořene (obr. 11) a zkoumat zda box daného uzlu koliduje s viditelnou oblastí (pomocí frustum cullingu, viz. kapitola 7.2), můžeme vyloučit najednou několik boxů. Zjistíme-li, že je uzel neviditelný, je automaticky neviditelné i celé pokračující větvení z tohoto uzlu.

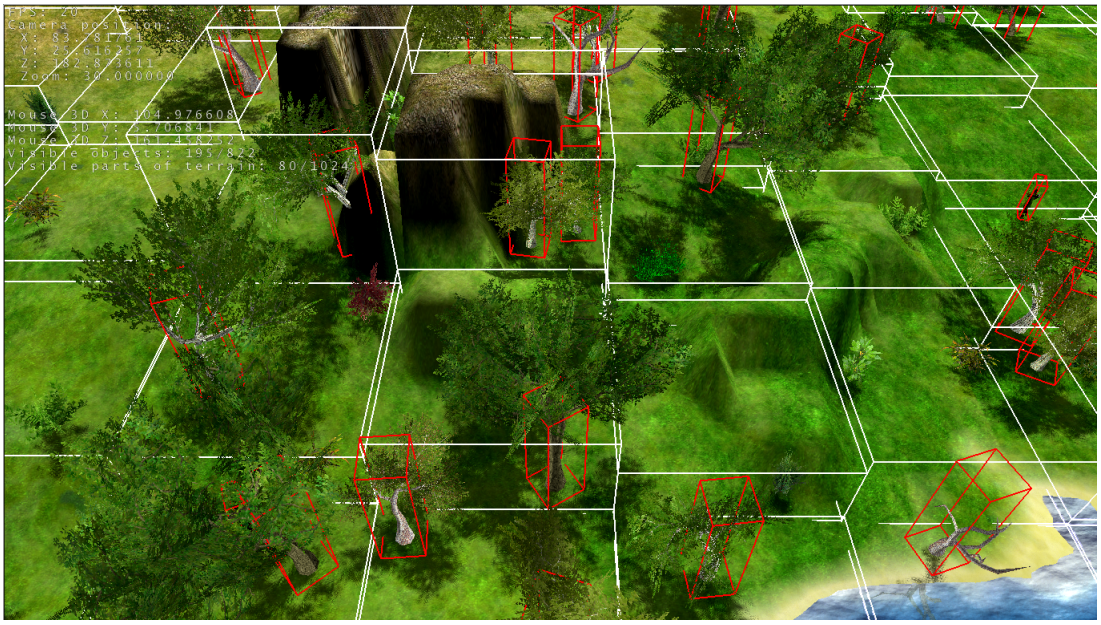


Obrázek 11: Průchod stromem

Implementace

V tomto enginu je quadtree naimplementován ve třídě *Box*. Voláním jejího konstruktoru dojde k vytvoření kořenového uzlu a dále také k novému volání konstruktorů čtyř dalších Boxů. Mezi parametry

konstruktoru Boxu patří mimo jiné proměnná *nesting* určující stupeň zanoření uzlu. Ta je při každém dalším zanoření zmenšena o jedna, pokud dosáhne nuly tak se jedná o listový uzel. Bylo testováno několik nastavení maximálního zanoření, nejoptimálnější se jeví zanoření 5. Větší už příliš zatěžovalo cpu nadměrným počtem kontrol kolizí a zpomalovalo tak plynulost hry. Menší zanoření při běžném zoomu postrádalo smysl.



Obrázek 12: Rozdělení mapy pomocí quadtree

Quadtree řeší také culling statických entit umístěných na mapu, pro pohyblivé entity není použit, muselo by docházet k periodické detekci ve kterém boxu se entita právě nachází (je pohyblivá), což z hlediska rychlosti není vhodným řešením. Třída *Box* obsahuje metodu *insertEntity*, pomocí které může být do stromu vložen nový objekt. Logika funkce prochází postupně stromem a porovnává boxy jednotlivých uzlů s bounding boxem nové entity. Po nalezení odpovídajícího uzlu je do něj uložen pointer na vkládanou entitu a také dojde k případné změně výšky boxu (je-li bounding box entity vyšší než box uzlu).

Vykreslení celé viditelné oblasti mapy je naimplementováno ve funkci *drawVisible*. Jednoduše dojde k průchodu stromem a volání *draw* funkcí všech obsažených objektů.

Aktualizace viditelnosti

Aby nedocházelo ke zbytečnému periodickému procházení quadtree i v případě, že nepohybujeme kamerou, je v enginu navržena přídavná struktura ukládající poslední viditelnou část mapy. Namespace *Octree* se samotnou třídou *Box* obsahuje také dynamické pole *visibleBoxes* ukládající pointery na poslední nalezené viditelné uzly. Flag *actualVisible* datového typu *bool* určuje, zda je sekvence ukazatelů ve *visibleBoxes* aktuální nebo ne. Pokud je tedy volána funkce *drawVisible*, je nejprve zkontrolováno *actualVisible* pokud je nastaveno na *true* nedochází k průchodu stromem, ale pouze k zavolání všech *draw* funkcí uložených uzlů. V opačném případě jsou nalezeny a uloženy nové viditelné uzly. Funkce *updateScene* je volána při jakémkoliv pohybu s kamerou. Dojde k nastavení *actualVisible* na *false* a vymazání pole *visibleBoxes*.

8 Závěr

V této práci bylo dosaženo všech vytyčených cílů a na vzdory prvním předpokladům se nám podařilo vytvořit fungující a hratelnou hru. Ukázkové demo zahrnuje více oblastí než dokumentuje tato práce. Dalším pozitivem je, že vývoj tohoto enginu bude pravděpodobně dále pokračovat a dojde také k rozšíření vývojového týmu o další programátory a grafiky. Poslední verze přiložená k této práci obsahuje:

- Grafický subsystém založený na OpenGL a SDL
- Detekci kolizí, založenou na použití bounding boxů, ale využívající také rozšířených mož-ností naimplementované logiky pro hledání cesty
- Správu událostí
- Herní logiku a AI, demo obsahuje několik budov, každá má své zaměření (např.: těžba dřeva, výběr daní, stavba dalších budov apod.), ale také příslušné jednotky co tyto služby reálně zajišťují.
- Generování herního prostředí
- GUI postavené na stromové struktuře, podobný princip jak je použit například v Qt
- Správu načítání: modelů, textur a shaderů
- Možnost skriptování pomocí LUA (zatím pouze GUI, do budoucna by mělo být rozšířeno na skriptování AI, tvorbu tzv. decision tree apod.)

Výsledný engine přidaný k této práci byl kompilován a testován na Windows, ale byla testována i jeho přenositelnost na Linux (konkrétně Ubuntu 10.04), kde funguje bez viditelných problémů. Mezi cíle do budoucna patří mimo jiné vytvoření komplexního a stabilního open source enginu, který bude do jisté míry univerzální a jednoduše rozšiřitelný hlavně díky skriptování. Velký důraz bude kladen také na jeho přenositelnost na jiné platformy.

Martin Dorazil

Literatura

[1] Gregory J., *"Game Engine Architecture"*, ISBN 9781568814131, 2009.

[2] Wright S. R., Haemel N., Sellers G., Lipchak B., *"OpenGL SuperBible Fifth Edittion"*, ISBN 9780321712615, 2010.

[3] Olsen J, *"Realtime Procedural Terrain Generation"*, University of Southern Denmark, 2004.

[4] Kessenich J., Baldwin D., Rost R., *"The OpenGL Shading Language 4.3"*, 2012.

[5] Reed N., *"Frustum And Occlusion Culling In A Quadtree-Based Terrain Renderer"*, 2004.

[6] McShaffry M., *"Game Coding Complete, Third Edition"*, ISBN 9781584506805, 2009.

[7] Dunn F., Parberry I., *"3D Math Primer for Granphics and Game Development"*, ISBN 9781556229114, 2002.

[8] McReynolds T., Blythe D., *"Advanced Graphics Programming Using OpenGL"*, ISBN 9781558606593, 2005.

[9] Dvořáková M., *"Kolizní systémy"*, Brno, 2007, Bakalářská práce na Fakultě informatiky Masarykovy Univerzity. Vedoucí práce doc. Ing. Jiří Sochor, CSc.

[10] Tobola P., *"Detekce kolizí těles"*, Brno, 2003, Masarykova Univerzita.

[11] Wikipedia.org, *"Quadtree"*, [ONLINE],
`http://en.wikipedia.org/wiki/Quadtree`

[12] Opengl.org, *"OpenGL"*, [ONLINE],
`http://www.opengl.org/`

[13] Opengl.org, *"Bounding volume"*, [ONLINE],
`http://en.wikipedia.org/wiki/Bounding_volume`

Přílohy

A Screenshot demo aplikace

Obrázek 13: Město vytvořené pomocí našeho enginu

B Budovy ve hře

Banka

- Budova vytvářející jednotky pro výběr daní
- Detekuje budovy ve svém okolí a posíla k nim své jednotky
- Získává zlato

Lovec

- Budova vytvářející jednotky pro lov zvěře
- Detekuje zvěř ve svém okolí a posíla k nim své jednotky
- Získává jídlo

Dřevorubec

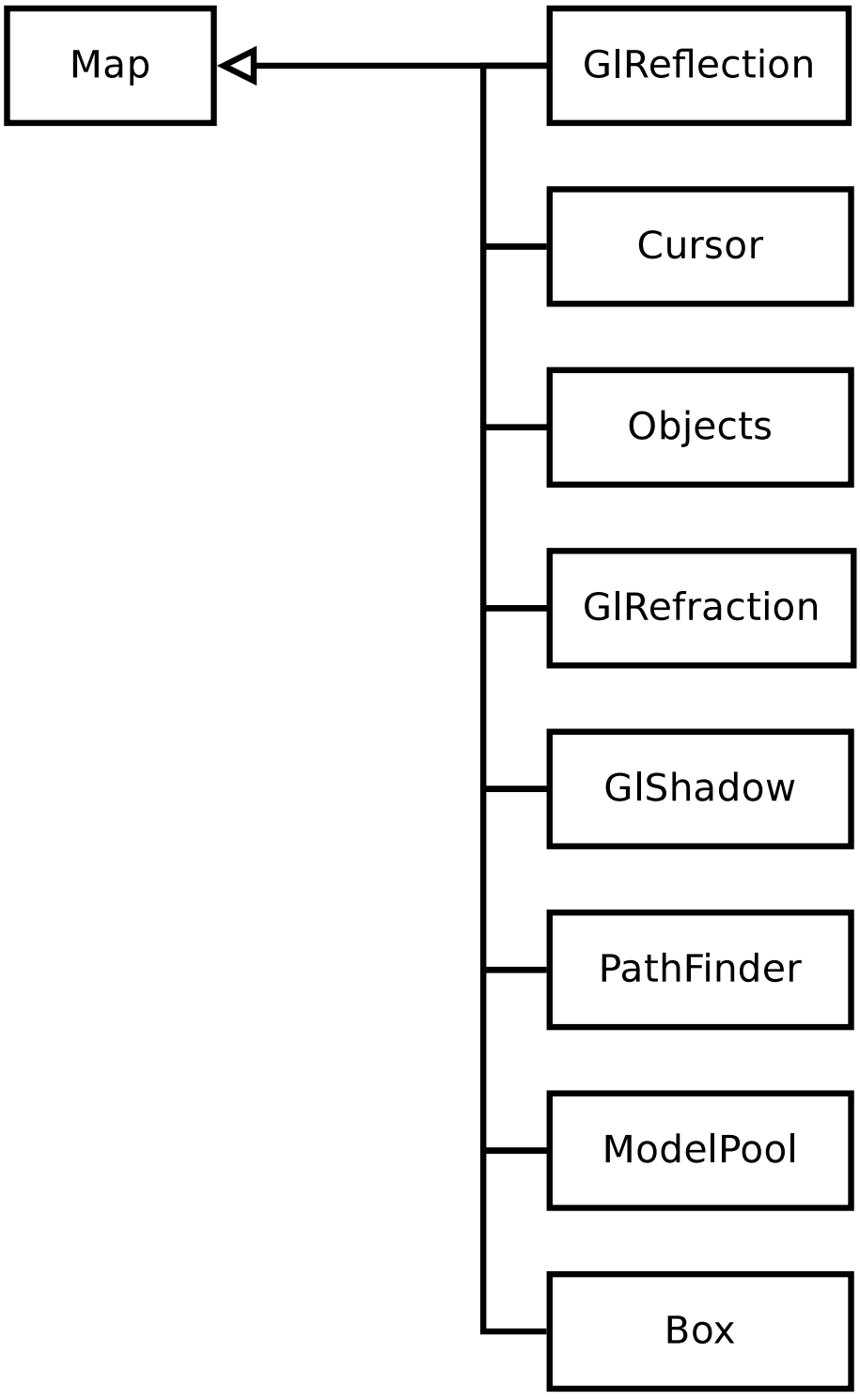
- Budova vytvářející jednotky pro kácení stromů
- Detekuje stromy ve svém okolí a posíla k nim své jednotky
- Získává dřevo

Hlavní budova

- Budova vytvářející jednotky pro stavbu jiných budov
- Detekuje stavby ve svém okolí a posíla k nim své jednotky

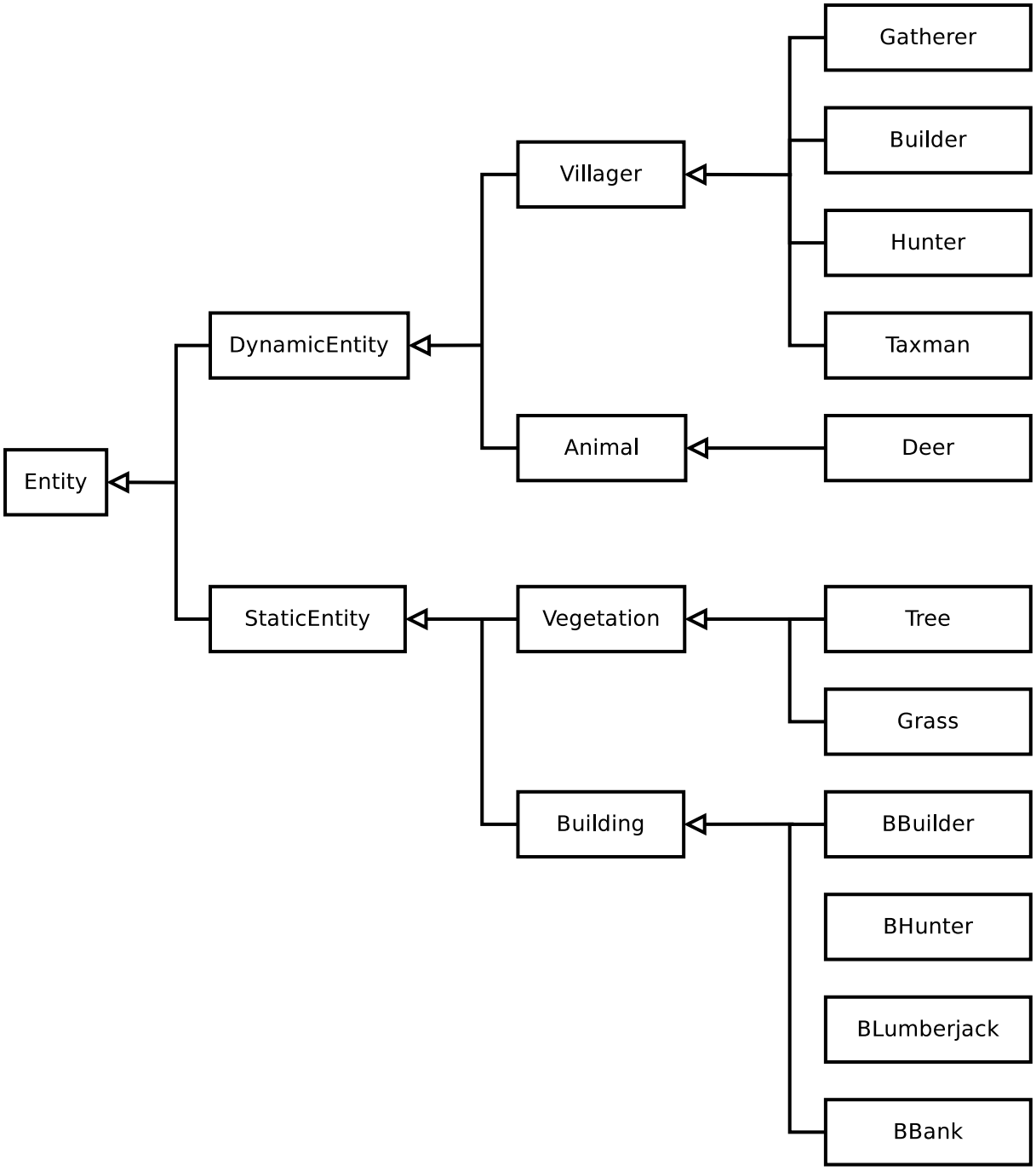
Obrázek 14: Seznam a popis budov použitých v demo aplikaci

C Třídní diagram mapy



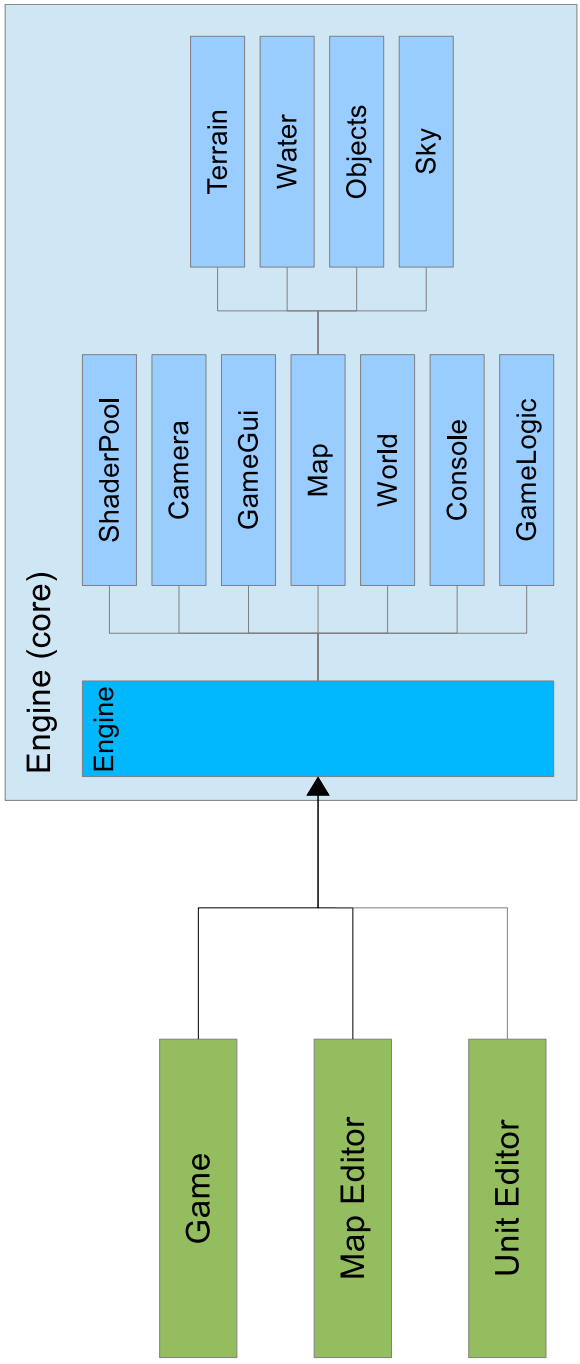
Obrázek 15: Třídní diagram vazeb mapy na ostatní části enginu

D Třídní diagram entit



Obrázek 16: Třídní diagram entit v enginu

E Návrh struktury enginu



Obrázek 17: Návrh struktury vysledného enginu s přidáním editoru map a jednotek